Introduction to Static Analysis

Using static analysis to find vulnerabilities at scale

Simon Gerst | 28. Juli 2022

Static Analysis vs. Dynamic Analysis

Static Analysis

- Analyzing programs *without* execution
- Accurate analysis is impossible[a] in general
- But we can use approximations 🎉

→ Trade-off: Accuracy vs. performance

[a] Halting problem and Rice's theorem :(

Static Analysis vs. Dynamic Analysis

Static Analysis

- Analyzing programs *without* execution
- Accurate analysis is impossible[a] in general
- But we can use approximations 🎉

→ Trade-off: Accuracy vs. performance

– Low(er) accuracy on big and complicated programs

\+ Can find vulnerabilities that dynamic analysis *cannot* find
E.g. bug is only present on rare configurations

[a] Halting problem and Rice's theorem :(

Dynamic Analysis

- Analyzing programs *during* execution
- (Usually) By observing a real or virtual CPU

Static Analysis vs. Dynamic Analysis

Static Analysis

- Analyzing programs *without* execution
- Accurate analysis is impossible[a] in general
- But we can use approximations 🎉

→ Trade-off: Accuracy vs. performance

– Low(er) accuracy on big and complicated programs

\+ Can find vulnerabilities that dynamic analysis *cannot* find
E.g. bug is only present on rare configurations

[a] Halting problem and Rice's theorem :(

Dynamic Analysis

- Analyzing programs *during* execution
- (Usually) By observing a real or virtual CPU

– Need a way to observe the program

\+ High accuracy

\+ Can find vulnerabilities that static analysis *cannot* find
E.g. *service A* writes to *file B* and another *service C* then reads *file B* unsafely, leading to RCE

Source-based vs. Binary-based

Source-based

- Source code needed
- Compilation-based:
  - Code needs to be compiled
  - CodeQL, Clang Static Analyzer, (Facebook) Infer
- No compilation needed:
  - Cppcheck, Joern, Semgrep

Source-based vs. Binary-based

Source-based

- Source code needed
- Compilation-based:
  - Code needs to be compiled
  - CodeQL, Clang Static Analyzer, (Facebook) Infer
- No compilation needed:
  - Cppcheck, Joern, Semgrep

Binary-based

- Works directly on a binary
- Joern
- Decompilation + source-based analyzer works somewhat[1][2]

[1] https://www.s3.eurecom.fr/docs/asiaccs22_mantovani.pdf
[2] https://security.humanativaspa.it/automating-binary-vulnerability-discovery-with-ghidra-and-semgrep/

CodeQL & Joern

CodeQL

- Developed by Semmle/GitHub
- All queries and most extractors are open source
- Evaluator is closed source :(
- C/C++, Javascript, Java, Kotlin[a], Python
- C#
- Ruby[b]
- Go
- Swift[a]

[a] Planned.
[b] Beta.

CodeQL & Joern

CodeQL

- Developed by Semmle/GitHub
- All queries and most extractors are open source
- Evaluator is closed source :(
- C/C++, Javascript, Java, Kotlin[a], Python
- C#
- Ruby[b]
- Go
- Swift[a]

[a] Planned.
[b] Beta.

Joern

- Developed by ShiftLeft
- All queries and all extractors are open source
- Evaluator is open source
- C/C++[a], Javascript[b], Java[b], Kotlin[b], Python[b]
- PHP[b]
- x86 assembly[b]
- Java bytecode[b]

[a] High maturity.
[b] Medium maturity.

Joern

Joern

- Based on code property graphs[a] (AST+CFG+PDG)
- First mentioned in 2016
- Domain specific "CPG query language"
- (C/C++): "uid should be changed before gid when dropping privileges"
- ```
  cpg
    .method("(?i)set(res|re|e|)uid")
    .callIn
    .whereNot(_.dominatedBy.isCall.name("set(res|re|e|)?gid"))
  ```

[a] https://comsecuris.com/papers/06956589.pdf
  ```

CodeQL

CodeQL

- Based on Datalog
- Logical, read-only, object-oriented and declarative (no side effects)
- First mentioned in 2007[a]
- Programming language + query engine and related tools
- (Java): "override equals and hashCode in classes"

```
from Class c
where c.declaresMethod("equals") and
  not(c.declaresMethod("hashCode")) and
  c.fromSource()
select c.getPackage(), c
```

[a] https://link.springer.com/chapter/10.1007/978-3-540-88643-3_3

CodeQL Use-Cases

Static analysis

- Write query for general bug class, e.g. XSS
- Run query against thousands of repositories
- Low false-positive rate wanted

CodeQL Use-Cases

Static analysis

- Write query for general bug class, e.g. XSS
- Run query against thousands of repositories
- Low false-positive rate wanted

Variant analysis

- You found a bug in a software

CodeQL Use-Cases

Static analysis

- Write query for general bug class, e.g. XSS
- Run query against thousands of repositories
- Low false-positive rate wanted

Variant analysis

- You found a bug in a software
- What if there is another bug with a similar cause?

CodeQL Use-Cases

Static analysis

- Write query for general bug class, e.g. XSS
- Run query against thousands of repositories
- Low false-positive rate wanted

Variant analysis

- You found a bug in a software
- What if there is another bug with a similar cause?
- Manually check all potential cases?

CodeQL Use-Cases

Static analysis

- Write query for general bug class, e.g. XSS
- Run query against thousands of repositories
- Low false-positive rate wanted

Variant analysis

- You found a bug in a software
- What if there is another bug with a similar cause?
- Manually check all potential cases?
- Very tedious → automate it
- Higher false-positive rate is acceptable

CodeQL Use-Cases

Static analysis

- Write query for general bug class, e.g. XSS
- Run query against thousands of repositories
- Low false-positive rate wanted

Variant analysis

- You found a bug in a software
- What if there is another bug with a similar cause?
- Manually check all potential cases?
- Very tedious → automate it
- Higher false-positive rate is acceptable
- Example: Hunting bugs in Accel-PPP with CodeQL[3]

[3] https://medium.com/csg-govtech/hunting-bugs-in-accel-ppp-with-codeql-8370e297e18f

CodeQL - Structure & Syntax

- SQL-like: Define *what* you want, not *how*

CodeQL - Structure & Syntax

- SQL-like: Define *what* you want, not *how*
- `import <language>` (where language is `java`, `javascript`, etc.)
- `from Class javaStringClass`
- `where javaStringClass.hasQualifiedName("java.lang", "String")`
- `select javaStringClass, "This␣is␣the␣Java␣String␣class."`

CodeQL - Structure & Syntax

- SQL-like: Define *what* you want, not *how*
- import <language> (where language is java, javascript, etc.)
- from Class javaStringClass
- where javaStringClass.hasQualifiedName("java.lang", "String")
- select javaStringClass, "This␣is␣the␣Java␣String␣class."
- Basically first-order logic:
- $< \forall | \exists > x.somePredicate(x) \equiv <$ forall | exists $>$(<Type> x | somePredicate(x))
- $< \wedge | \vee | \rightarrow | \neg > \equiv <$ and | or | implies | not $>$
- $[1, 10] \equiv$ [1..10] (inclusive range from 1 to 10)
- $=$, is the equality operator
- _, is the don't care value
- x instanceof <Type> $\equiv$ holds if x is of type <Type>

CodeQL - Types

- boolean: true and false
- float: 64-bit (!) floating point numbers, such as 6.28 and -0.618
- int: 32-bit two's complement integers, such as -1 and 42
- string: Finite strings of 16-bit characters
- Custom classes: Class, Method, MethodAccess, ...

CodeQL - Predicates

- Create reusable logic and give it a name

Predicates without result

```
predicate isSmallEvenNumber(int i) {
  i % 2 = 0 and // is even?
  i in [1..10] // is small?
}

from int i
where isSmallEvenNumber(i)
select i, "is even."
```

CodeQL - Predicates

- Create reusable logic and give it a name

Predicates without result

```
predicate isSmallEvenNumber(int i) {
  i % 2 = 0 and // is even?
  i in [1..10] // is small?
}

from int i
where isSmallEvenNumber(i)
select i, "is even."
```

Query result

i	message
2	is even.
4	is even.
6	is even.
8	is even.
10	is even.

CodeQL - Predicates With Result

Predicates with result

```
string getCreator(string language) {
  language = "Java" and result = "Sun"
  or
  language = "Rust" and result = "Mozilla"
  or
  language = "C#" and result = "Microsoft"
}

from string creator
where getCreator(_) = creator and
  creator.prefix(1) = "M"
select creator, "Creator starts with 'M'."
```

CodeQL - Predicates With Result

Predicates with result

```
string getCreator(string language) {
  language = "Java" and result = "Sun"
  or
  language = "Rust" and result = "Mozilla"
  or
  language = "C#" and result = "Microsoft"
}

from string creator
where getCreator(_) = creator and
  creator.prefix(1) = "M"
select creator, "Creator starts with 'M'."
```

Query result

creator	message
Microsoft	Creator starts with 'M'.
Mozilla	Creator starts with 'M'.

CodeQL - Predicates With Result

Predicates with result

```
string getCreator(string language) {
  language = "Java" and result = "Sun"
  or
  language = "Rust" and result = "Mozilla"
  or
  language = "C#" and result = "Microsoft"
}

from string creator
where getCreator(_) = creator and
  creator.prefix(1) = "M"
select creator, "Creator starts with 'M'."
```

Query result

creator	message
Microsoft	Creator starts with 'M'.
Mozilla	Creator starts with 'M'.

Syntactic sugar

<type> name(<t1> var1, ...) (with explicit result)

≡

predicate name(<type> result, <t1> var1, ...)

(desugared)

CodeQL - Transitive Closure

Sample Java class hierarchy

```
class Rectangle extends Shape {}
class Square extends Rectangle {}
class RedRectangle extends Rectangle {}
class Circle extends Shape {}
```

CodeQL - Transitive Closure

Sample Java class hierarchy

```
class Rectangle extends Shape {}
class Square extends Rectangle {}
class RedRectangle extends Rectangle {}
class Circle extends Shape {}
```

hasSupertype relation

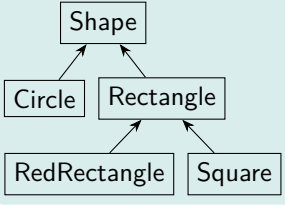
- *a hasSupertype b* ⇔ a has b as a supertype
- Rectangle *hasSupertype* Shape
- Square *hasSupertype* Shape
- RedRectangle *hasSupertype* Rectangle
- Circle *hasSupertype* Shape

CodeQL - Transitive Closure

Sample Java class hierarchy

```
class Rectangle extends Shape {}
class Square extends Rectangle {}
class RedRectangle extends Rectangle {}
class Circle extends Shape {}
```

hasSupertype relation as a graph

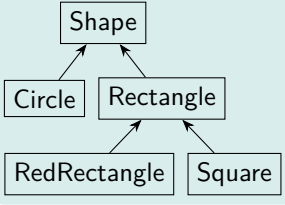


CodeQL - Transitive Closure

Sample Java class hierarchy

```
class Rectangle extends Shape {}
class Square extends Rectangle {}
class RedRectangle extends Rectangle {}
class Circle extends Shape {}
```

hasSupertype relation as a graph



- How to find classes that extend Shape?

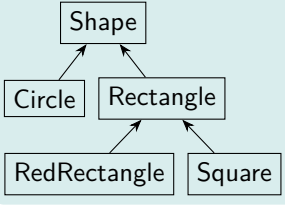
→ Find all c such that c *hasSupertype* Shape holds

CodeQL - Transitive Closure

Sample Java class hierarchy

```
class Rectangle extends Shape {}
class Square extends Rectangle {}
class RedRectangle extends Rectangle {}
class Circle extends Shape {}
```

hasSupertype relation as a graph



- How to find classes that extend Shape?

→ Find all c such that c *hasSupertype* Shape holds

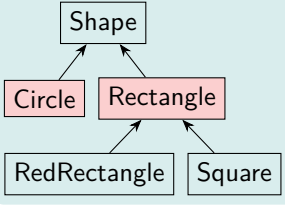
```
from Class c, Class shapeClass
where c.getASupertype() = shapeClass and
  shapeClass.hasName("Shape")
select c
```

CodeQL - Transitive Closure

Sample Java class hierarchy

```
class Rectangle extends Shape {}
class Square extends Rectangle {}
class RedRectangle extends Rectangle {}
class Circle extends Shape {}
```

hasSupertype relation as a graph



- How to find classes that extend Shape?

→ Find all c such that c *hasSupertype* Shape holds

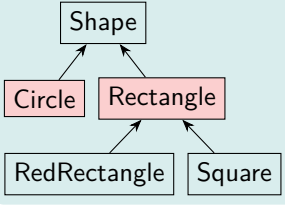
```
from Class c, Class shapeClass
where c.getASupertype() = shapeClass and
  shapeClass.hasName("Shape")
select c
```

CodeQL - Transitive Closure

Sample Java class hierarchy

```
class Rectangle extends Shape {}
class Square extends Rectangle {}
class RedRectangle extends Rectangle {}
class Circle extends Shape {}
```

hasSupertype relation as a graph



- How to find classes that extend Shape?

→ Find all c such that c *hasSupertype* Shape holds

```
from Class c, Class shapeClass
where c.getASupertype() = shapeClass and
  shapeClass.hasName("Shape")
select c
```

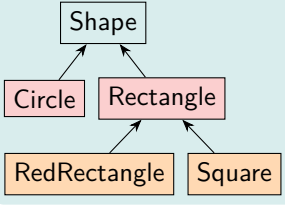
- How to also find classes that transitively extend Shape?

CodeQL - Transitive Closure

Sample Java class hierarchy

```
class Rectangle extends Shape {}
class Square extends Rectangle {}
class RedRectangle extends Rectangle {}
class Circle extends Shape {}
```

hasSupertype relation as a graph



- How to find classes that extend `Shape`?

→ Find all `c` such that `c` *hasSupertype* `Shape` holds

```
from Class c, Class shapeClass
where c.getASupertype+() = shapeClass and
  shapeClass.hasName("Shape")
select c
```

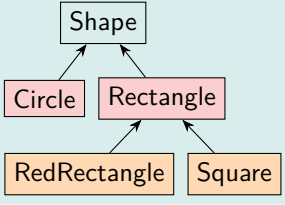
- How to also find classes that transitively extend `Shape`?
- `getASupertype+()` ≡ transitive closure („all paths with one or more steps“)

CodeQL - Transitive Closure

Sample Java class hierarchy

```
class Rectangle extends Shape {}
class Square extends Rectangle {}
class RedRectangle extends Rectangle {}
class Circle extends Shape {}
```

hasSupertype relation as a graph



- How to find classes that extend `Shape`?

→ Find all `c` such that `c` *hasSupertype* `Shape` holds

```
from Class c, Class shapeClass
where c.getASupertype+() = shapeClass and
  shapeClass.hasName("Shape")
select c
```

- How to also find classes that transitively extend `Shape`?
- `getASupertype+()` ≡ transitive closure („all paths with one or more steps“)
- `getASupertype*()` ≡ transitive reflexive closure („all paths with zero or more steps“)
- $+$ and $*$ can be applied to any binary predicate

CodeQL - Recursion

- Predicates can recursively call other predicates

Counting from 0 to 10

```
int getANumber() {
  result = 0
  or
  result <= 10 and result = getANumber() + 1
}
```

CodeQL - Recursion

- Predicates can recursively call other predicates

Counting from 0 to 10

```
int getANumber() {
  result = 0
  or
  result <= 10 and result = getANumber() + 1
}
```

- getASupertype+() can be rewritten using recursion

Rewriting transitive closures

```
Class getASupertypeTransitive() {
  result = this.getASupertype()
  or
  result = this.getASupT().getASupTTrans()
}
```

CodeQL - Classes

- Describe a set of values

→ Easy reuse

Without using classes

```
from Class c, Class shapeClass
where c.getASupertype+() = shapeClass and
  shapeClass.hasName("Shape")
select c
```

Rewritten using classes

```
class Shape extends Class {
  Shape() {
    this.hasName("Shape")
  }
}


from Class c
where c.getASupertype+() instanceof Shape
select c
```

CodeQL - Classes 2

Characteristic predicates

```
class Foo extends Bar {
  Foo() { ... }
}
```

- `Foo() { ... }` is the *characteristic* predicate of class `Foo`
- `Foo` is the set of all values for which the characteristic predicate holds

CodeQL - Classes 2

Characteristic predicates

```
class Foo extends Bar {
  Foo() { ... }
}
```

- `Foo() { ... }` is the *characteristic* predicate of class Foo
- `Foo` is the set of all values for which the characteristic predicate holds

Multiple inheritance

```
class SpecialNumber extends OddNumber,
                            PowerOfThreeNumber {
}
```

- SpecialNumber is the intersection of OddNumber and PowerOfThreeNumber
- SpecialNumber is the set of all values for which the characteristic predicate of OddNumber holds and the characteristic predicate of PowerOfThreeNumber holds

CodeQL - Casts

Casts

- Allow constraining the type of an expression

CodeQL - Casts

Casts

- Allow constraining the type of an expression
- Postfix style cast: `x.(Foo)` restricts the type of `x` to `Foo`

CodeQL - Casts

Casts

- Allow constraining the type of an expression
- Postfix style cast: `x.(Foo)` restricts the type of `x` to `Foo`
- Prefix style cast: `(Foo)x` also restricts the type of `x` to `Foo`

CodeQL - Casts

Casts

- Allow constraining the type of an expression
- Postfix style cast: `x.(Foo)` restricts the type of `x` to `Foo`
- Prefix style cast: `(Foo)x` also restricts the type of `x` to `Foo`
- `x.(Foo)` is exactly equivalent to `((Foo)x)`
- Prefix style casts are practically never used!

CodeQL - Casts

Casts

- Allow constraining the type of an expression
- Postfix style cast: `x.(Foo)` restricts the type of `x` to `Foo`
- Prefix style cast: `(Foo)x` also restricts the type of `x` to `Foo`
- `x.(Foo)` is exactly equivalent to `((Foo)x)`
- Prefix style casts are practically never used!
- Don't know to what type `x` can be cast?

→ `x.getAPrimaryQlClass()` gets the name of a primary (most precise) QL class of `x` (only use for debugging!)

CodeQL - Casts

Casts

- Allow constraining the type of an expression
- Postfix style cast: `x.(Foo)` restricts the type of `x` to `Foo`
- Prefix style cast: `(Foo)x` also restricts the type of `x` to `Foo`
- `x.(Foo)` is exactly equivalent to `((Foo)x)`
- Prefix style casts are practically never used!
- Don't know to what type `x` can be cast?

→ `x.getAPrimaryQlClass()` gets the name of a primary (most precise) QL class of `x` (only use for debugging!)

Most specific type

```
public int midpoint(int low, int high) {
        return (low + high) / 2;
}
```

- How to find all `return`s that are a division by two?

```
from ReturnStmt retStmt, Expr retVal
where retStmt.getResult() = retVal
select retVal
```

CodeQL - Casts

Casts

- Allow constraining the type of an expression
- Postfix style cast: `x.(Foo)` restricts the type of `x` to `Foo`
- Prefix style cast: `(Foo)x` also restricts the type of `x` to `Foo`
- `x.(Foo)` is exactly equivalent to `((Foo)x)`
- Prefix style casts are practically never used!
- Don't know to what type `x` can be cast?

→ `x.getAPrimaryQlClass()` gets the name of a primary (most precise) QL class of `x` (only use for debugging!)

Most specific type

```
public int midpoint(int low, int high) {
        return (low + high) / 2;
}
```

- How to find all `return`s that are a division by two?

```
from ReturnStmt retStmt, Expr retVal
where retStmt.getResult() = retVal
select retVal
```

- `getResult` returns an expression, can we cast this to a division?

CodeQL - Casts

Casts

- Allow constraining the type of an expression
- Postfix style cast: `x.(Foo)` restricts the type of `x` to `Foo`
- Prefix style cast: `(Foo)x` also restricts the type of `x` to `Foo`
- `x.(Foo)` is exactly equivalent to `((Foo)x)`
- Prefix style casts are practically never used!
- Don't know to what type `x` can be cast?

→ `x.getAPrimaryQlClass()` gets the name of a primary (most precise) QL class of `x` (only use for debugging!)

Most specific type

```
public int midpoint(int low, int high) {
        return (low + high) / 2;
}
```

- How to find all `return`s that are a division by two?

```
from ReturnStmt retStmt, Expr retVal
where retStmt.getResult() = retVal
select retVal, retVal.getAPrimaryQlClass()
```

- `getResult` returns an expression, can we cast this to a division?

CodeQL - Casts

Casts

- Allow constraining the type of an expression
- Postfix style cast: `x.(Foo)` restricts the type of `x` to `Foo`
- Prefix style cast: `(Foo)x` also restricts the type of `x` to `Foo`
- `x.(Foo)` is exactly equivalent to `((Foo)x)`
- Prefix style casts are practically never used!
- Don't know to what type `x` can be cast?

→ `x.getAPrimaryQlClass()` gets the name of a primary (most precise) QL class of `x` (only use for debugging!)

Most specific type

```
public int midpoint(int low, int high) {
        return (low + high) / 2;
}
```

- How to find all `return`s that are a division by two?

```
from ReturnStmt retStmt, Expr retVal
where retStmt.getResult() = retVal
select retVal, retVal.getAPrimaryQlClass()
```

- Yes! We get `DivExpr` as a result, so we can cast to it

CodeQL - Casts

Casts

- Allow constraining the type of an expression
- Postfix style cast: `x.(Foo)` restricts the type of `x` to `Foo`
- Prefix style cast: `(Foo)x` also restricts the type of `x` to `Foo`
- `x.(Foo)` is exactly equivalent to `((Foo)x)`
- Prefix style casts are practically never used!
- Don't know to what type `x` can be cast?

→ `x.getAPrimaryQlClass()` gets the name of a primary (most precise) QL class of `x` (only use for debugging!)

Most specific type

```
public int midpoint(int low, int high) {
        return (low + high) / 2;
}
```

- How to find all `return`s that are a division by two?

```
from ReturnStmt retStmt, DivExpr retVal
where
  retStmt.getResult() = retVal and
  retVal.getRightOperand()
  .(IntegerLiteral).getIntValue() = 2
select retVal
```

CodeQL - Writing Queries

lgtm.com

- Written in „Query console“ (basic editor with auto-complete)
- Run directly against projects (no account needed)
- Aggregate projects to lists and run against hundreds of projects (needs account)
- Some very large projects may be unavailable and can not be queried → VSCode + Plugin

CodeQL - Writing Queries

lgtm.com

- Written in „Query console“ (basic editor with auto-complete)
- Run directly against projects (no account needed)
- Aggregate projects to lists and run against hundreds of projects (needs account)
- Some very large projects may be unavailable and can not be queried → VSCode + Plugin

VSCode + Plugin

- VSCode + CodeQL extension for Visual Studio Code
- Recommended to use the starter workspace[4]
- Databases can be downloaded[5] from lgtm.com and imported
- Can use `codeql` binary to manually build databases for projects unavailable on lgtm.com

[4] https://github.com/github/vscode-codeql-starter
[5] E.g. this database for PowerShell https://lgtm.com/projects/g/PowerShell/PowerShell/ci/#ql

Finding Vulnerabilities at Scale

- Pretty simple process:

1. Find interesting vulnerability pattern, write new query
2. Run query against hundreds of projects using the `lgtm.com` platform
3. Many false-positives? If so, refine query and goto 2
4. Else, find security contact for project
5. Report and get CVE if needed (E.g. no CVE needed for personal test projects)

Finding Vulnerabilities at Scale

- Pretty simple process:

1. Find interesting vulnerability pattern, write new query
2. Run query against hundreds of projects using the `lgtm.com` platform
3. Many false-positives? If so, refine query and goto 2
4. Else, find security contact for project
5. Report and get CVE if needed (E.g. no CVE needed for personal test projects)

Time (very rough *guess*timate)

- Query writing: 10 hours
- Reviewing initial query results and refinements: 3-5 hours
- Finding security contacts, writing reports and getting CVEs: 10-20 hours

→ Contacting and reporting can take *far* longer than writing!
90-day deadline + unresponsive vendors = pain

GitHub Security Lab Bounty Program[6]

All for one, one for all

- Write a query that models a new vulnerability class (that is not already modeled)
- Find at least one CVE that the query covers – CVE is not mandatory to be new/discovered by you
- The more severe CVEs, the better
- Awards of up to $6000 USD can be granted

[6] https://securitylab.github.com/bounties/

GitHub Security Lab Bounty Program[6]

All for one, one for all

- Write a query that models a new vulnerability class (that is not already modeled)
- Find at least one CVE that the query covers – CVE is not mandatory to be new/discovered by you
- The more severe CVEs, the better
- Awards of up to $6000 USD can be granted

The Bug Slayer

- Use your previously written query to find and fix vulnerabilities
- Find at least four new vulnerabilities of high severity, or two new vulnerabilities of critical severity
- Awards of up to $7,800 USD for multiple critical CVEs can be granted

[6] https://securitylab.github.com/bounties/

Resources

- https://codeql.github.com/docs/codeql-language-guides/abstract-syntax-tree-classes-for-working-with-java-programs/
- https://codeql.github.com/docs/codeql-language-guides/
- https://codeql.github.com/docs/codeql-language-guides/codeql-for-java/
- https://codeql.github.com/docs/writing-codeql-queries/ql-tutorials/
- https://codeql.github.com/docs/writing-codeql-queries/codeql-queries/
- https://jorgectf.gitlab.io/blog/post/practical-codeql-introduction/
- https://help.semmle.com/QL/ql-support/ql-training/
- https://intrigus.org/research/2021/08/05/finding-insecure-jwt-signature-validation-with-codeql/ (shameless plug)

Questions?

Ping `intrigus` on Slack

or

DM `@intrigus_` on twitter (note the underscore)

or

open a discussion at

https://github.com/github/codeql/discussions